OPENCL EMBEDDED PROFILE PROTOTYPE IN MOBILE DEVICE

Jyrki Leskelä, Jarmo Nikula, Mika Salmela
Devices R&D, Renewal Projects
Nokia Corporation

ABSTRACT

Abstract—Programmable Graphics Processing Unit (GPU) has over the years become an integral part of today's computing systems. The GPU use-cases have gradually been extended from graphics towards a wide range of applications. Since the programmable GPU is now making its way to mobile devices, it is interesting to study these new use-cases also there. To test this, we created a programming environment based on the embedded profile of the fresh Khronos OpenCL standard and ran it against an image processing workload in a mobile device with CPU and GPU back-ends. The early results on performance and energy consumption with CPU+GPU configuration were promising but also suggest there is room for optimization.

Index Terms— OpenCL, Image processing, Parallel processing, Embedded systems, Computer graphics hardware.

1. INTRODUCTION

The explosion of peak processing power of the desktop Graphics Processing Units (GPU) has progressed into a situation where typical GPU is up to 40-60 times faster than CPU in certain algorithms [1]. This performance advantage has been combined with programmability via new graphics API versions such as Direct3D® 10 [2] and OpenGL™ 3.0 [3], through manufacturer specific programming techniques such as CUDA™ [4] and through start-ups such as RapidMind [5]. As a result, the desktop GPU is now used for applications such as scientific computing, industrial and financial modeling, and computer vision, yielding real savings in hardware costs and energy consumption.

GPUs are also emerging as embedded versions into mobile devices, such as smartphones. The first of them were fixed-function accelerators [6], whereas the latest standard, OpenGL ES 2.0 [7], allows for programmatic access to the GPU via programs written in a shading language. Applications such as mobile video editing [8] and mobile computer vision [9] are examples that have suitable computing use cases for current and future generations of mobile GPU. Usually there is no heavy 3D graphics running concurrently with these applications so the GPU is available for other processing.

A recent standard computing technique for heterogeneous set of multiprocessors, OpenCL™ [10], allows run-time allocation of workloads between CPU, GPU, and other accelerators. OpenCL has also a profile for embedded devices.

This study reports the results of running an example image processing algorithm with a prototype implementation of OpenCL embedded profile running on TI OMAP™ 3430 Application Processor [11]. The algorithm kernel is able to run concurrently on CPU, GPU and DSP, though DSP was not included in this study. The rest of the paper is organized as follows: Section 2 gives an overview of the features of the OpenCL and its embedded profile. Section 3 presents the hardware used in the study. Section 4 defines the OpenCL programming environment prototype. Section 5 specifies the example image processing algorithm and Section 6 reports the main performance and energy measurements with CPU and GPU configurations. Section 7 summarizes the conclusions.

2. OPENCL AND EMBEDDED PROFILE

OpenCL is a new C-based parallel programming model for GPUs, CPUs, and other processors. It provides well-defined numerical accuracy (IEEE 754 rounding with specified max error), online or offline compilation and building of compute kernel executables and a rich set of built-in functions. Device specifics are abstracted at low level in OpenCL APIs and kernel language.

The two main APIs of OpenCL 1.0 are:

- Platform Layer API provides a hardware abstraction layer to query, select, and initialize compute devices, and create compute contexts and work queues.
- Runtime API provides means to execute compute kernels in selected devices, and to manage scheduling, compute, and memory resources.

The data is passed between the host and compute devices as buffer or image object. Buffer objects are unstructured 1D buffers such as C-arrays – they can contain scalar types, vector types, as well as user-defined structures. Inside kernel programs they are accessed via pointers. Image objects represent 2D or 3D textures, frame-buffers or images. They must be addressed through built-in functions. Sampler objects are related to images, since they describe how to sample the image (select the mode of addressing and filtering).

The computing phase of OpenCL is performed by queuing kernel functions for execution over a specified one-, two-, or three-dimensional index space called NDRange (N-Dimensional Range), see Fig. 1. The index space can be divided at intermediate level into work-groups. The single execution of a kernel against one of the indexes is called a work-item. The index space values are typically used in the kernel to guide the addressing of input and output data. To allow efficient data parallelism, memory

consistency between work-items is not guaranteed except at barriers and other synchronization points.

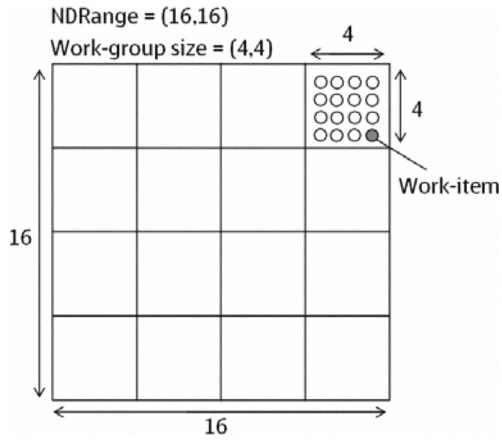

Figure 1. Example of OpenCL index space (2D).

OpenCL kernels are represented as strings defined with a C-based syntax. The main additions to C include:

- Vector data types
- Rich set of data-type conversion functions
- Image types
- Function qualifier __kernel to declare kernels
- Functions to query work-item identifiers
- Address space qualifiers for kernel arguments
- Image read/write functions using sampler objects
- Synchronization (barriers and memory fences)

There are also extensions to the core OpenCL 1.0 specification. Both Full Profile and Embedded Profile implementations can use these extensions:

- Double precision floating-point types and functions
- Atomic functions
- 3D Image writes
- Byte addressable stores
- Built-in functions to support half types

2.1. Embedded Profile Restrictions

The OpenCL 1.0 Embedded Profile is defined to be a subset of OpenCL rather than a separate specification. Embedded Profile can be identified via a macro from the kernel language and through parameters of platform and device info in the OpenCL API. Its requirements are more relaxed than the full profile:

- Online compiler is optional for embedded devices
- Vectors/scalars of 64-bit integers are not supported
- 3D image support is optional
- The requirements of 2D images are relaxed to the level of OpenGL ES 2.0 textures
- The minimum memory/object size requirements as well as requirements on floating point types are scaled down

3. RESEARH HARDWARE ENVIRONMENT

This study was done with Texas Instruments (TI) OMAP™ 3430 Application Processor [11] containing ARM Cortex™ A8 CPU [13], TI TMS320C64x DSP, PowerVR™ SGX 530 GPU [15] and shared DDR SDRAM. This chipset is not optimized for OpenCL but it is quite representative as a recent heterogeneous mobile device application processor. The OMAP™ 3430 was run in an openly available Zoom Mobile Development Kit (MDK) [12].

4. OPENCL EMBEDDED PROFILE PROTOTYPE

The OpenCL Embedded Profile prototype under study was comprised of the host platform running in ARM CPU and several compute device back-ends, see Fig 2.

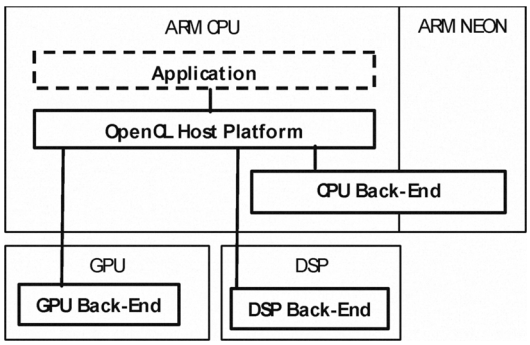

Figure 2. Structure of OpenCL prototype.

The feature scope of the prototype is the core specification of OpenCL 1.0 Embedded Profile without extensions. As of writing, some features are not yet implemented. The main limitations are:

- Synchronization primitives are not functional
- Conformance testing not yet completed
  - → Known issues in kernel language
  - → Some API functions still empty
- No native floating point in DSP back-end
  - → Software floating point library
- GPU back-end layered on top of OpenGL ES 2.0
  - → No free-form buffer support
  - → Features/data types in kernel language

The following software package was used in the underlying platform:

- CPU: Open Source Linux TI BSP 12.20/23.8 (based on Linux Kernel 2.6.24)
- GPU: Imagination Technology drivers for SGX 2008_09_08
- DSP: Compiler cl6x 6.0.16, Bridge 2.6.29

The CPU host program was compiled without optimizations but the CPU back-end was compiled with gcc -03 level optimizations for ARM NEON instruction set, and it also had several assembler level optimizations for OpenCL specific language functionality. The GPU back-end was compiled into a CPU program containing an OpenGL ES 2.0 shader program as a string. The string was compiled into GPU executable in run-time.

5. EXAMPLE IMAGE PROCESSING APPLICATION

Figure 3 presents the structure of the example image processing application from OpenCL point-of-view. It follows the general usage principles of OpenCL API.

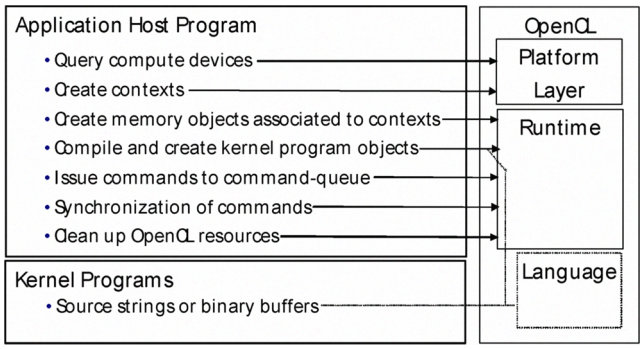

Figure 3. OpenCL application structure.

5.1. Example Algorithm

The example algorithm consists of several image processing steps for a three channel image merged into a single image processing kernel. As shown in Fig. 4, the first step is a geometry correction algorithm for the correction of e.g. barrel or pincushion distortion of camera lens. The second step is a 3x3 one-pass square linear spatial filter approximating a Gaussian Blur operation. The third step is an adjustment of the image color saturation and hue. After these steps, the result image is returned for further processing.

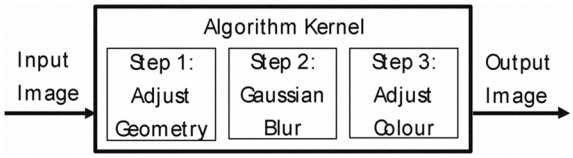

Figure 4. Design of the example algorithm.

The detailed design is given in equations (1), (2) and (3). Geometry was adjusted with constants zoom (***Z***) and bend (***B***) as follows

$$g(x,y)=f(x_c+(x-x_c)r^B Z, y_c+(y-y_c)r^B Z),$$
$$r=\sqrt{(\frac{x-x_c}{\frac{1}{2}W})^2+(\frac{y-y_c}{\frac{1}{2}H})^2}\quad , \qquad (1)$$

where ***W***, ***H*** are image width and height and (x_c, y_c) is the correction center-point; values $x_c = W/2$ and $y_c = H/2$ were used. The equation can be reduced by shifting the square root into the bending constant ***B*** ($\sqrt{a}^b = a^{\frac{1}{2}b}$).

Gaussian blur was approximated with

$$g(x,y)=\sum_{s=-1}^{1}\sum_{t=-1}^{1} w(s,t) f(x+s, y+t), \qquad (2)$$

where w(s, t) contains the filtering constants. The computation was simplified further in implementation using the symmetry over x and y axis.

The color adjustment was done as follows:

$$g(x,y)=Cf(x,y), C=C_{RGB}M_{HS}C_{YUV}. \qquad (3)$$

The adjustment matrix M_{HS} was a constant because the hue and saturation adjustment remained constant over the full image so it was possible to calculate the full 3x3 conversion matrix ***C*** in advance.

5.2. OpenCL Kernel Programs

Listing 1 shows the OpenCL kernel which is processing floating point scalars and vectors. Image2d_t type was used for input and output buffers instead of free-form buffers due to the GPU back-end limitations. Thus, coordinates need to be relative scaled for the range 0...1. Some coefficient data was calculated beforehand in the host program to gain speedup.

Listing 1:

```
// imageProcessingTest.cl
__kernel void imageProcessingTest(
  float bend,
  float zoom,
  float4 adjust_r, // Conversion matrix
column 1
  float4 adjust_g, // Conversion matrix
column 2
  float4 adjust_b, // Conversion matrix
column 3
  float4 gauss_coeff,
  __read_only image2d_t srcimage,
  __write_only image2d_t destimage,
  int width,
  int height)
{
const sampler_t sampler =
  CLK_NORMALIZED_COORDS_TRUE |
  CLK_ADDRESS_CLAMP_TO_EDGE |
  CLK_FILTER_NEAREST;

float var_t_4;
float hypo;
float2 size;
float2 xy;
float2 mycoord;
int2 destPos;

destPos.x = get_global_id( 0 ); // x index
destPos.y = get_global_id( 1 ); // y index

// Coordinate adjustment
mycoord.x = (float)(destPos.x) /
(float)(width);
mycoord.y = (float)(destPos.y) /
(float)(height);
float2 center;
center.x = 0.5f; // W / 2
center.y = 0.5f; // H / 2
xy = mycoord - center;
xy = xy / center;
hypo = dot(xy, xy);
var_t_4 = pow(hypo, bend);
var_t_4 = var_t_4 * zoom;
xy = xy * var_t_4;
xy = xy * center;
xy = xy + center;
```

```
// Read a 3x3 gauss blurred image
float2 south; // Works only with conservative
bend
float2 east;
south.x = 0.0f;
south.y = zoom / (float)(height);
east.x = zoom / (float)(width);
east.y = 0.0f;
float4 srcColor =
  read_imagef(srcimage, sampler, xy) *
  gauss_coeff.x;
srcColor += (
  read_imagef(srcimage, sampler, xy+south) +
  read_imagef(srcimage, sampler, xy-south) +
  read_imagef(srcimage, sampler, xy+east) +
  read_imagef(srcimage, sampler, xy-east)
  ) * gauss_coeff.y;
srcColor += (
  read_imagef(srcimage, sampler,
xy+south+east) +
  read_imagef(srcimage, sampler, xy-south-
east) +
  read_imagef(srcimage, sampler, xy+south-
east) +
  read_imagef(srcimage, sampler, xy-
south+east)
  ) * gauss_coeff.z;

// Adjustment of contrast and saturation
float4 adjColor;
adjColor.x = dot (adjust_r, srcColor);
adjColor.y = dot (adjust_g, srcColor);
adjColor.z = dot (adjust_b, srcColor);
adjColor.w = srcColor.w;

// Saturate and write image
srcColor = clamp(adjColor, 0.0f, 1.0f);
write_imagef(destimage, destPos, srcColor);
}
```

The pow() function was known to be crucial in CPU environment, thus the very slow standard version was replaced with a fast approximate version of it. After that adjustment, the relative time consumed per frame by the pow() was approximately 5% for the CPU and 3.5% for the GPU. The difference can be considered to be small enough in order not to skew the results too much.

It was considered unfair to require CPU to use image2d_t type for the data buffer since image2d_t is specifically designed to support GPU native hardware-implemented texture formats. With CPU it requires expensive software-emulation. To overcome this, we conducted the CPU test runs with a variant of the kernel that used OpenCL memory buffers instead of image2d_t. The resulted CPU version consumed only 23% of the original, image2d_t emulated time. It had a slightly more lightweight edge clamping compared to image2d_t version, which explains the greater variance of min and max processing time of the frames in CPU.

6. MEASUREMENTS

The Zoom MDK CPU speed was configured to 550 MHz, the GPU speed to 110 MHz and the SDRAM speed to 166 MHz during the tests. The DSP tests were not yet completed in an openly available environment thus they are not included.

The measurement was conducted for 100 32-bit RGBA frames each of size 2016 x 1512. All the other parameters remained constant, but the zoom level (***Z***) was changed for every frame. In case of combined CPU+GPU run, the scheduling was done frame-by-frame by placing each frame to the processors in the order they became ready to accept new frames, leading to out-of-order completion of the frames.

6.1. Performance Measurements

Table 1 shows general timing information for the processed frames. More detailed timing of the GPU's 2.4 second frame time showed that 0.3 seconds is due to transferring and setting up input data for GPU and 0.4 seconds is still consumed when reading back the result data. Thus, roughly only 1.7 seconds is left for pure execution and 0.7 seconds is due to data transfers forth and back.

Table 1. Test run time consumptions.

100 Frames 32b-RGBA 2016x1512	Alone		Combined	
	CPU	GPU	CPU	GPU
Frames/device	100	100	19	81
Execute time (s)	864.9	240.4	237.4	247.5
Average s/frame	**8.6**	**2.4**	12.5	3.1
			2.5	
Min s/frame	6.98	2.33	10.87	2.39
Max s/frame	10.94	2.49	12.62	3.16

The main finding in this performance measurement is that GPU was clearly faster than CPU. Detailed investigation pointed out that the processing of floating point numbers, especially vectors, is much faster with GPU than with the performance-compromised VFPLite unit on the Cortex-A8 CPU [13]. Instead, ARM's NEON unit should perform fine also with floating point vectors, but due to compiler and our OpenCL implementation limitations the instruction set is not fully utilized. However, according to our experience, benefitting on NEON requires more pipelined vector computations than the tested algorithm really has; otherwise the performance boost is easily lost on NEON register conversions. Probably an integer approximated CPU version with optimized inner loop would have competed with GPU much better. On the other hand, not all the opportunities of GPU were used either – for example the OpenGL ES vertex shader was not fully utilized by the OpenCL backend. The workload would have been even more suitable for the GPU, if bilinear filtering mode would have been used.

The total time consumed for the test run with CPU and GPU simultaneously was 257.3 seconds. Figure 5 illustrates the detailed timing of the frames, time progressing from left to right. Red blocks illustrate the time used to compute a frame (the execute time in the table) and the black/light bars between the frames illustrate idle/queue time. The long idle times between the CPU frames are caused by non-optimal scheduling of the current version of the OpenCL prototype. Actually, our implementation kept CPU idle if it finished during execution of GPU kernel. Also, GPU stays idle at the end while CPU is computing the last frame.

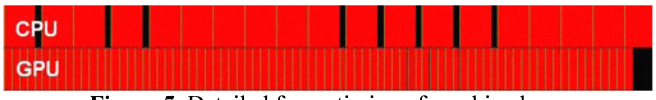

Figure 5. Detailed frame timing of combined run.

Contrary to expectations that processing some load in CPU in parallel with GPU would speed up the total processing time, it wasn't the case. Strangely, processing 81 frames with GPU in this case took longer than all 100 frames when GPU was run alone. The reason is that data transferring to and from GPU is mostly done by CPU: processing same time other frame on CPU slows down data transfers. Measurements showed that data transfer times roughly doubled from GPU alone case, becoming 0.6 s for setting the input data and 0.8 s for getting back the result. As the total frame time was 3.1 s, this means that execution time kept in original 1.7 seconds, indicating that memory bandwidth isn't the bottle-neck. Thus, GPU just stays idle longer than when it is run alone, when waiting data transfers to complete. On the other hand, frame computation done by CPU meanwhile cannot compensate that loss and we don't achieve speed-up at all. However, the result depends on relative time share between data transfers and computation, and evidently, a different algorithm having more computation per frame would gain some speed-up.

6.2. Possible Scheduling Improvements

Since memory bandwidth is not the limiting factor, a simple improvement for CPU+GPU case would be to run GPU thread with higher priority. This would enable data transfers to complete in original time, 0.7 s per frame. Thus, selecting 84 frames for GPU processing would give the optimum as follows: Total time becomes 84 × 2.4 s = 202 s, including 84 × 0.7 s = 59 s CPU time due to data transfers. This leaves 202 s – 59 s = 143 s to be used for CPU's frame computation, being enough for processing 143 s / 8.6 s = 16 complete frames. That way, theoretically all 84 + 16 = 100 frames could be processed in only 202 seconds.

It should also be possible to asynchronously overlap data transfers with GPU execution, using double-buffering. This would eliminate the transfer times in GPU-only case, except the first and last transfers, thus shrinking time to 100 × 1.7 s + 0.7 s ≈ 171 s for the examined test run. Further, if this is applied for CPU+GPU case, optimal throughput could be achieved with the scheduling shown in Figure 6, where F0...F4 refer to first five frames and 'get'/'set' refer to data transfers forth and back.

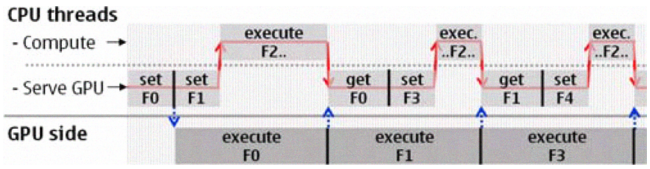

Figure 6. GPU-priority scheduling using double-buffering.

Assuming memory bandwidth is still sufficient, the test case would then optimally run in roughly 154 seconds, when 90 frames are computed using GPU and the rest 10 using CPU (GPU time becomes 90 × 1.7 s + 0.7 s ≈ 154 s and after subtracting data transfers, 154 s – 90 × 0.7 s = 91 s is left for actual CPU execution, being enough for 91 s / 8.6 s ≈ 10 frames). The run time would thus be 40% less than with the measured original. Our intention is to create an improved version to allow this kind of scheduling. However, optimal workload sharing gets different if speed-balance changes favoring CPU, until at some point it might be best to ignore GPU in order to maximize CPU utilization for execution. Generally, scheduling that takes into consideration relative speed-differences and cross-dependencies (such as data-transfers) between processors needs more investigation.

6.3. Energy Measurements

Another interesting study topic was energy consumption. For that purpose, the battery energy consumption was measured for CPU-only, GPU-only and CPU+GPU configurations. Power supply voltage was set to 3.8V. Average idle current was measured for each run separately, being between 515 mA and 550 mA. That was considered as offset current and is already subtracted from the measured values shown in Table 2.

Table 2. Results of energy measurements.

Efficacy	CPU alone	GPU alone	CPU+ GPU
Frames (n)	4	20	20
Time (s)	43.5	48.6	52.1
Current (mA)	95	61	132
Power (mW)	361	232	502
Energy/frame (J)	**3.93**	**0.56**	**1.31**

GPU uses only 0.56 J / 3.93 J = 14% of the energy needed by CPU for a single frame computation. More precise analyzing of the accurately sampled power consumption curve showed that if data transfer periods are excluded, GPU consumed only 40 mA during kernel execution. For an average 1.7 second pure execution time this means only 3.8 V × 40 mA × 1.7 s = 0.26 J consumed per frame. Thus, more than half of the GPU-alone joules shown in the table are due to data transfers. Also, in case of CPU+GPU together, joules-per-frame becomes higher, as can be expected.

The main explaining factor for GPU advantage is the different processor architecture: GPU is originally designed for parallel bulk data processing, whereas CPU is oriented for running complex programs with narrower sequential data flow. OpenCL computing is designed for bulk data processing, thus preferring GPU architecture. Though both architectures support single-instruction-multiple-data (SIMD) vector operations, only GPU implements higher level of parallelism, single-program-multiple-data (SPMD), on HW level. Other explaining things are GPU's better floating point computation compared to our incomplete utilization of the CPU's NEON unit.

Also, GPU's lower MHz (110 vs. 550) as such is an advantage. Operating frequency has implications to temperature and power dissipation increases more rapidly than just linearly to frequency [15]. As a result, for low-power data processing it is generally better to choose design having low MHz with high parallelism than vice-versa, high MHz with little parallelism.

7. CONCLUSIONS

The study initially showed some benefits achievable with OpenCL in the mobile device environment. Offloading some computations to an embedded GPU can with some algorithms provide both good

speedups and energy efficiency. Further study is required for more complex algorithms, integer oriented workloads, and offloading some processing to other accelerators such as DSP. Also, the current prototype implementation needs to be optimized towards higher performance and more complete functionality. The mentioned improvement possibilities in CPU-GPU scheduling will be examined next.

Run-time workload allocation for a heterogeneous set of processors with OpenCL brings flexibility to achieve algorithm performance needs across various system designs. The future evolution of mobile device processor implementations will greatly depend on the system design trade offs (energy consumption, silicon size and cost vs. performance) between the processing units. GPU has originally architecture that cleanly supports OpenCL approach, making it also easy to add more parallel execution units and get them automatically utilized. This is also both cost- and energy-efficient, since all surrounding circuitry doesn't need to be multiplied. Instead, having more parallelism in CPU is conventionally achieved by multiplying the whole core, as recent dual- and quad-core announcements have shown. This is expensive and energy-inefficient, and still making use of these additional cores requires changes in software side, too. OpenCL opens whole new door also for CPU designers and unification of the two different worlds may become possible.

8. REFERENCES

[1] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E., Stone, and J.C Phillips, "GPU Computing", *Proceedings of the IEEE, Volume 96, Issue 5*, IEEE, USA, pp. 879 – 899, May 2008.

[2] *Programming Guide for Direct3D 10*, Microsoft Corporation, msdn.microsoft.com, USA, 2009.

[3] M. Segal, K. Akeley (ed.), *The OpenGL Graphics System: A Specification (Version 3.0 - August 11, 2008)*, www.khronos.org, Khronos Group, , USA, August 2008.

[4] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide; Version 2.0*, nVidia Inc, www.nvidia.com, July 2008.

[5] M. Monteye, *Rapidmind White Paper: Rapidmind Multi-Core Development Platform*, Rapidmind Inc., www.rapidmind.com, February 2008.

[6] K., Pulli, T., Aarnio, V., Miettinen, K., Roimela, and J., Vaarala, *Mobile 3D Graphics with OpenGL ES and M3G*, Morgan Kauffman, USA, 2007.

[7] A. Munshi, D. Ginsburg, and D. Shreiner, *OpenGL ES 2.0 Programming Guide*, Addison-Wesley, USA, July 2008.

[8] A. Houruranta, A. Islam, and F. Chebil, "Video and Audio Editing for Mobile Applications", *IEEE International Conference on Multimedia & Expo*, IEEE, Canada, pp. 1305-1038, 2006.

[9] G. Takacs, V. Chandrasekhar, N. Gelfand, Y. Xiong, W.-C. Chen, T. Bismpigiannis, R. Grzeszczuk, K. Pulli, and B. Girod, "Outdoor Augmented Reality on Mobile Phone using Loxel-Based Visual Feature Organization", *ACM International Conference on Multimedia Information Retrieval,* ACM, Canada, pp. 427-434, October 2008.

[10] Khronos OpenCL Working Group, A. Munshi Ed., *The OpenCL Specification, Version 1.0, Rev. 43*, Khronos Group, USA, May 2009.

[11] *OMAP3 family of multimedia application processors*, Texas Instruments Inc., http://focus.ti.com, 2007.

[12] *Product Brief: Zoom Omap34x Mobile Development Kit*, Logic Product Development Inc, http://www.logicpd.com, 2008.

[13] *White Paper: Architecture and Implementation of the ARM Cortex-A8 Microprocessor*, ARM Ltd., http://www.arm.com, October 2005.

[14] *Fact Sheet: SGX Graphics IP Core Family*, Imagination Technologies Ltd., November 2007.

[15] Vojin G. Oklobdzija, *The Computer Engineering Handbook*, CRC Press, 2002. ISBN 0849308852, 9780849308857.